Python Lists and Tuples

Main Points

1 Creating Lists

There are many data types in *Python*. Two specific data types that are used quite often and can hold multiple objects are *lists* and *tuples*.

- Lists can be used to store different object types within a single structure, such as...
  - Integers
  - Floats
  - Strings
  - Lists
  - Many more
- Define a list by surrounding the objects with square brackets `[ ]` and separating the objects with commas, i.e. `[1, 2, 3, 4.5, "knights", [5, 6, 7]]`
- Most lists will contain one object type instead of several
  - Numeric values; floats and/or integers
  - Strings
- Empty lists are created by...
  - Using an empty set of square brackets, i.e. `empty_list = []`
  - Using the `list()` function, i.e. `empty_list = list()`
- Lists are mutable
  - Items in list can be changed (replaced actually)
  - Items can be added or removed
  - The order of the list can be modified

A list with integers, floats, strings, and expressions

```
my_list = [42, 6.8, 2.78, 'homework', 1001, 3.14, 1/3, abs(-100), -25]
print(my_list)
```

```
[42, 6.8, 2.78, 'homework', 1001, 3.14, 0.3333333333333333, 100, -25]
```

2 Creating Tuples

Tuples are similar to lists. They are constructed using parentheses instead of square brackets; i.e. `my_tuple = (1, 2, 3)`. However, tuples are immutable.

- Tuples can contain the same mix of data types as lists
- Tuples can be created without parentheses, but parentheses are preferred for clarity
- Used in functions that return more than one value
- Used when creating (x, y) and (x, y, z) vectors
- Convert lists (and some other data types) into tuples using the `tuple()` function
- Tuples can be created with a single item, i.e. `my_tuple = (4,)`
- Immutable; cannot change, add or remove items in tuples; must recreate instead

Tuple creation

```
a_tuple = (2, 5, 42, 3.14, 99.999, 'hello', 9%2)
b_tuple = 4, 5, 6     # look mom, no parentheses
print(a_tuple)
print(b_tuple)
```

```
(2, 5, 42, 3.14, 99.999, 'hello', 1)
(4, 5, 6)
```

Assigning values to a tuple from a function that returns multiple values

```
(quotient, remainder) = divmod(7, 2)
print(f"Quotient of 7 \u00f7 2 = {quotient}")
print(f"Remainder of 7 \u00f7 2 = {remainder}")
```

```
Quotient of 7 ÷ 2 = 3
Remainder of 7 ÷ 2 = 1
```

3 The `range()` and `list()` Functions

Lists of evenly spaced values are used very often, especially in engineering problems. The `range()` function creates an *iterable* of evenly spaced values that can be used like a list. The `list()` function can be used to convert a range object into an actual list if needed.

- `range(start, end, step)` accepts up to three integer arguments...
    - Start value
    - End value
    - Step size
- Creates what is referred to in mathematics as a half-open range
    - The start value is included in the range
    - The end value is not included in the range
- Values in ranges are always integers
- One argument is used: `range(end)`
    - The range includes all integers from `0` to `end - 1`
    - `range(10)` creates an iterable of all integers from 0 to 9, inclusive
- Two arguments are used: `range(start, end)`...
    - The range includes all integers from `start` to `end - 1`
    - `range(4, 8)` includes all integers from 4 to 7, inclusive
- Three arguments are used: `range(start, end, step)`...
    - The range includes all integers from `start` to `end - 1` in steps of `step`
    - `range(3, 10, 2)` includes all odd integers from 3 to 9, inclusive
    - `end` must be greater than `start` for positive steps
    - `end` must be less than `start` for negative steps
- A range can be converted into a list using the `list()` function
- `list(range(10))` creates the list `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

Range with one argument

```
range_1 = range(8)
print(range_1)       # prints the range object, not the the values in it
print(list(range_1))
```

```
range(0, 8)
[0, 1, 2, 3, 4, 5, 6, 7]
```

Range with two arguments

```
range_2 = range(5, 11)
print(list(range_2))
```

```
[5, 6, 7, 8, 9, 10]
```

Range with three arguments and positive step

```
range_3 = range(1, 11, 3)
print(list(range_3))
```

```
[1, 4, 7, 10]
```

Range with three arguments and negative step

```
range_4 = list(range(10, -1, -2))
print(list(range_4))
```

```
[10, 8, 6, 4, 2, 0]
```

4 List Functions

- `len()` returns the number of objects in a list, i.e. its length
  - Counts only "first level" objects: lists and tuples within the list each count as one object
  - `len([0, 1, 2, 3, 4])` will return 5
  - `len([0, 1, 2, 3, [4, 5, 6]])` will return 5
- `sum()` – the sum of all values in the list; must be numeric
- `min()` and `max()` – the smallest or largest value in the list
  - Must be either all strings or all numeric values
  - If numeric values the results are based on magnitude
  - If strings the results are based on alphabetical order
- `del(x)` will completely delete a list (or any other object)

Using `len()`, `min()`, and `max()` on a numeric list

```
number_list = [6, 67, 12, 3.14, 90, -20, 2.78, 1/2, 1e3, 5e-2, 67]
print(number_list)
print(f"number_list has {len(number_list)} objects")
print(f"The sum of the values in number_list is {sum(number_list)}")
print(f"The minimum value in number_list is {min(number_list)}")
print(f"The maximum value in number_list is {max(number_list)}")
print(f"The average is {sum(number_list)/len(number_list)}")
```

```
[6, 67, 12, 3.14, 90, -20, 2.78, 0.5, 1000.0, 0.05, 67]
number_list has 11 objects
The sum of the values in number_list is 1228.47
The minimum value in number_list is -20
The maximum value in number_list is 1000.0
The average is 111.67909090909092
```

Using `min()` and `max()` on a list of strings

```
fruits = ['orange', 'watermelon', 'apple', 'tomato', 'kiwi']
print(fruits)
print(f"The minimum fruit is {min(fruits)}")
print(f"The maximum fruit is {max(fruits)}")
```

```
['orange', 'watermelon', 'apple', 'tomato', 'kiwi']
The minimum fruit is apple
The maximum fruit is watermelon
```

Deleting a list

```
>>> number_list = [6, 67, 12, 3.14, 90, -20, 2.78, 1/2, 1e3, 5e-2, 67]
>>> del(number_list)
>>> number_list
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'number_list' is not defined
```

5 Accessing List Items: Indexing and Slicing

- *List indexing* works the same as string indexing
- Lists are zero indexed and square brackets are used to specify the index value
- `my_list[3]` returns the object in the 3rd index position (4th item)
- `my_list[-1]` returns the last object in the list

List indexing diagram

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | <-- from the left; jennys_list[3] = 5
|===|===|===|===|===|===|===|
| 8 | 6 | 7 | 5 | 3 | 0 | 9 | <-- jennys_list
|===|===|===|===|===|===|===|
|-7 |-6 |-5 |-4 |-3 |-2 |-1 | <-- from the right; jennys_list[-3] = 3
```

Indexing a list

```
>>> jennys_list = [8, 6, 7, 5, 3, 0, 9]

>>> jennys_list[3]
5
>>> jennys_list[-3]
3
```

- *List slicing* works just like string slicing
- `my_list[0:3]`, `my_list[:3]`, and `my_list[0:3:1]` return the same slices
- Tuples are indexed and sliced exactly the same as lists and strings

List slicing diagram

```
 0   1   2   3   4   5   6   7 <-- from the left
 |===|===|===|===|===|===|===|
 | 6 | 0 | 6 | 0 | 8 | 4 | 2 | <-- good_time
 |===|===|===|===|===|===|===|
-7  -6  -5  -4  -3  -2  -1   | <-- from the right with positive step
 |   |   |   |   |   |   |   |
-8  -7  -6  -5  -4  -3  -2  -1 <-- from the right with negative step

good_time = [6, 0, 6, 0, 8, 4, 2]
print(f"good_time[1:4] = {good_time[1:4]}")
print(f"good_time[-5:-2] = {good_time[-5:-2]}")
print(f"good_time[-3:-7:-1] = {good_time[-3:-7:-1]}")
```

```
good_time[1:4] = [0, 6, 0]
good_time[-5:-2] = [6, 0, 8]
good_time[-3:-7:-1] = [8, 0, 6, 0]
```

Some examples of indexing and slicing a list of nubmers.

Index position 9 (aka 10th value when starting with 1)

```
>>> number_list = [6, 42, 12, 3.14, 90, -20, 2.78, 1/2, 1e3, 5e-2, 67]
>>> number_list[9]
0.05
```

Second to the last value using negative indexing

```
>>> number_list[-2]
0.05
```

The 3rd to 7th indices, inclusive

```
>>> number_list[3:8]
[3.14, 90, -20, 2.78, 0.5]
```

The last 4 values

```
>>> number_list[-4:]
[0.5, 1000.0, 0.05, 67]
```

Every other value

```
>>> number_list[::2]
[6, 12, 90, 2.78, 1000.0, 67]
```

All of the values without any numbers in the slice

```
>>> number_list[:]
[6, 42, 12, 3.14, 90, -20, 2.78, 0.5, 1000.0, 0.05, 67]
```

6 Changing List Items

- Change a single item by assigning a new value to a specific index of the list
  - `my_list[4] = 12` changes the value in the 4th index position to 12
  - `my_list[-1] = 10` changes the last value in the list to 10
- Change multiple items using the list slicing syntax
  - The assigned object must be list, even if it only has a single value
  - The assigned object does not need to be the same length as the slice
  - `my_list[:5] = [9, 8, 7, 6, 5]` changes the values of the first 5 items
  - `my_list[2:5] = [100]` will replace three values with a single value

Modifying a single value using indexing

```
print(number_list)
number_list[7] = 42
print(number_list)
```

```
[6, 67, 12, 3.14, 90, -20, 2.78, 0.5, 1000.0, 0.05, 67]
[6, 67, 12, 3.14, 90, -20, 2.78, 42, 1000.0, 0.05, 67]
```

Modifying the last three values by slicing from the right

```
number_list[-3:] = [8, 6, 7]
print(number_list)
```

```
[6, 67, 12, 3.14, 90, -20, 2.78, 42, 8, 6, 7]
```

Replacing the first five values with a single value using a slice

```
number_list[:5] = [99999]
print(number_list)
```

```
[99999, -20, 2.78, 42, 8, 6, 7]
```

7 List Methods

There are list methods related to modifying lists, counting, searching, and changing the order of the objects in lists. Since lists are mutable, most of the methods change the original list instead of making a copy.

Printing just the list methods

```
>>> print([item for item in dir(list) if "__" not in item])
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
 ↪  'remove', 'reverse', 'sort']
```

7.1 Methods for Adding and Removing List Objects

- `list.append()` adds a single object to the end of list, even if that object is a list
- `list.extend()` extends a list with the items from another list or iterable
- `list.insert(index, object)`
    - Inserts `object` into the `index` position of the list
    - All items right of `index` move to the right
    - `my_list.insert(3, 999)` inserts 999 into index positon 3
- `list.clear()` deletes all items in a list, leaving it empty
- `list.remove(value)`
    - Removes the first occurrence of `value`
    - Results in an error if the `value` is not in the list
- `list.pop()`
    - Removes and return the value from a specific index position
    - With no argument, last item is removed and returned
    - With an argument, the the value at the index position provided is removed and returned

The following code cells show examples of the above list methods using the same list. The list will change at each step.

Append a list with a single value

```
my_list = [1, 2, 3]
print(my_list)
my_list.append(1000)
print(my_list)
```

```
[1, 2, 3]
[1, 2, 3, 1000]
```

Append a list with a list

```
my_list.append([7, 8, 9])
print(my_list)
```

```
[1, 2, 3, 1000, [7, 8, 9]]
```

Extend a list with three values using `range()`

```
my_list.extend(range(98, 101))
print(my_list)
```

```
[1, 2, 3, 1000, [7, 8, 9], 98, 99, 100]
```

Extend a list with a string

```
my_list.extend("hello!")
print(my_list)
```

```
[1, 2, 3, 1000, [7, 8, 9], 98, 99, 100, 'h', 'e', 'l', 'l', 'o', '!']
```

Insert a value into the list

```
my_list.insert(5, 42)
print(my_list)
```

```
[1, 2, 3, 1000, [7, 8, 9], 42, 98, 99, 100, 'h', 'e', 'l', 'l', 'o', '!']
```

Pop the last value and assign it to a name

```
a = my_list.pop()
print(a)
print(my_list)
```

```
!
[1, 2, 3, 1000, [7, 8, 9], 42, 98, 99, 100, 'h', 'e', 'l', 'l', 'o']
```

Pop a specific value from the list without assigning to a name

```
my_list.pop(0)
print(my_list)
```

```
[2, 3, 1000, [7, 8, 9], 42, 98, 99, 100, 'h', 'e', 'l', 'l', 'o']
```

Remove a specific value from the list after checking if its in the list

```
if 1000 in my_list:
    my_list.remove(1000)
print(my_list)
```

```
[2, 3, [7, 8, 9], 42, 98, 99, 100, 'h', 'e', 'l', 'l', 'o']
```

Clear a list

```
my_list.clear()
print(my_list)
```

```
[]
```

7.2 Methods for Counting, Searching, and Changing Order

- `list.count(value)` counts the number of times `value` appears in the list
- `list.index(value)` finds the object `value` in a list
  - If found, the index position of the first occurrence of `value` is returned
  - If not found, an error is generated
  - Optional arguments
    - Starting index
    - Ending index
  - It's a good idea to check if a value is in a list before using `.index`
- `list.sort()` will sort a list in ascending order by default
  - The list must contain either all numeric or all string values
  - Including the optional argument `reverse=True` will sort the list in descending order
- `list.reverse()` will flip the order of the list
- The `sorted()` function is like `list.sort()` except it makes a sorted *copy* of the list
  - List is sorted in ascending order by default
  - Add the optional argument `reverse=True` for descending order
- The `reversed()` function makes flipped *copy* of the list as an iterator, not a list

Counting values in a list

```
>>> print(number_list)
[6, 42, 12, 3.14, 90, -20, 2.78, 0.5, 1000.0, 0.05, 67]

>>> number_list.count(5)   # how many occurrences of 5
0
>>> number_list.count(6)   # how many occurrences of 6
1
```

Finding the value 42 in the list

```
>>> number_list.index(42)
1
```

Searching for a value that is not in the list – error

```
>>> # where is 1001 (hint: there isn't one)
>>> number_list.index(1001)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 1001 is not in list

>>> 1001 in number_list    # better to test if value is in list first
False
```

Reverse the order of a list

```
>>> number_list.reverse()     # does not display the changed list
>>> print(number_list)        # must print it to see the change
[67, 0.05, 1000.0, 0.5, 2.78, -20, 90, 3.14, 12, 42, 6]
```

Sort in ascending order

```
>>> number_list.sort()
>>> print(number_list)
[-20, 0.05, 0.5, 2.78, 3.14, 6, 12, 42, 67, 90, 1000.0]
```

Sort in descending order

```
>>> number_list.sort(reverse=True)
>>> print(number_list)
[1000.0, 90, 67, 42, 12, 6, 3.14, 2.78, 0.5, 0.05, -20]
```

Create a sorted copy of a list

```
>>> second_list = sorted(number_list)
>>> print(second_list)
[-20, 0.05, 0.5, 2.78, 3.14, 6, 12, 42, 67, 90, 1000.0]
>>> print(number_list)
[1000.0, 90, 67, 42, 12, 6, 3.14, 2.78, 0.5, 0.05, -20]
```

Create a reversed copy (not actually a list though)

```
>>> reversed(number_list)
<list_reverseiterator object at 0x7f96bbf6d700>

>>> list(reversed(number_list))  # convert the reversed object to a list
[-20, 0.05, 0.5, 2.78, 3.14, 6, 12, 42, 67, 90, 1000.0]
```

8 The `zip()` Function and Making Tables from Lists

Often there are lists of values/data that would be nice to display together in tabular form. Printing a list or multiple lists does not make a nicely displayed table on its own. The key to a good looking table lies in using the built-in `zip()` function and the `tabulate` module.

- The function `zip()` connects two or more lists together much like the teeth in a zipper
- Zipping creates a list of tuples where each tuple contains one value from each list being zipped
- The `zip()` function creates a special iterable object that can be converted to list
- To print a `zip` object and see the values it contains requires first converting it to a list

Zipping together two lists

```
>>> zipped = zip([1, 2, 3, 4, 5], [6, 7, 8, 9, 10])

>>> list(zipped)
[(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
```

- A 3rd party module named `tabulate` can make tables from zipped values
- Inside the module is the `tabulate()` function that needs to be passed to the `print()` function
- Visit https://bitbucket.org/astanin/python-tabulate for a complete overview of `tabulate`
- Some options...
  - `tablefmt=` options include `"simple"`, `"grid"`, and `"plain"`
  - Row alignment can be accomplished with `numalign="center"` or `numalign="left"`
  - The argument `floatfmt=(".1f",".2f")` will show...
    - The first column with one-decimal place
    - The second with two-decimal places
    - Values in columns must be floats for this to work

The following example creates two lists named `list_a` and `list_b`. The `range()` function is used to create `list_a` with the values `0, 10, 20, 30, 40, 50`. The second list, `list_b`, is filled with five random integers between 0 and 100. The two lists are zipped together to create a simple table with the headings `tens` and `random` with the numbers centered.

Printing a table with tabulate

```
import random
from tabulate import tabulate
list_a = range(1, 51, 10)
list_b = random.sample(range(101), 5)
zipped_list = zip(list_a, list_b)
headers = ['tens', 'random']
print(tabulate(zipped_list, headers,
               numalign='center', tablefmt='fancy'))
```

```
 tens    random
------  --------
  1       28
  11      29
  21      3
  31      85
  41      40
```

9 List Related String Methods

- The string method `str.split()` can easily turn a sentence or phrase into a list of words
  - Without an argument, `.split()` will split the string at spaces
  - With an argument, the string will be split at every location the argument string is found
- The string method `str.join(list)` can join together a list of strings to create a single string
  - Acts on a string that “connects” the strings from the list (often a single space `" "`)
  - The argument is required and must be a list of strings
  - `" ".join(my_string_list)` will create a string whose “words” are separated by spaces

Splitting and joining

```
my_string = "And now for something completely different"
my_string_list = my_string.split()
print(my_string_list)

new_string = " ".join(my_string_list)
print(new_string)
```

```
['And', 'now', 'for', 'something', 'completely', 'different']
And now for something completely different
```

More splitting and joining

```
another_string = "1,2,3,4,5,6"
number_string_list = another_string.split(",")
print(number_string_list)
new_list = "---".join(number_string_list)
print(new_list)
```

```
['1', '2', '3', '4', '5', '6']
1---2---3---4---5---6
```

The following example splits a string to create a list. Then the last item in the list is changed to a different string before joining the list back into a string.

Even more splitting and joining

```
name = 'My name is Brian'
print(name)
name_list = name.split()
print(name_list)
name_list[-1] = 'Mr. Brady'
name = " ".join(name_list)
print(name)
```

```
My name is Brian
['My', 'name', 'is', 'Brian']
My name is Mr. Brady
```

Using the string method `.replace()` to do the same thing

```
name = 'My name is Brian'
name = name.replace("Brian", "Mr. Brady")
print(name)
```

```
My name is Mr. Brady
```

10 Some Creative Commons Reference Sources for This Material

- emphThink Python 2nd Edition Allen Downey, chapter 10
- emphThe Coder's Apprentice Pieter Spronck, chapters 11 and 12
- emphA Practical Introduction to Python Programming Brian Heinold, chapters 7 and 8
- emphAlgorithmic Problem Solving with Python John Schneider, Shira Broschat, and Jess Dahmen, chapters 6 and 7